Constraint Checking with Partial Information
(Extended Abstract)

Ashish Gupta
Yehoshua Sagiv†
Jeffrey D. Ullman
Jennifer Widom

Dept. of Computer Science
Stanford Univ., Stanford CA 94305

email: {agupta,sagiv,ullman,widom}@cs.stanford.edu
contact phone: (415) 723–9745
FAX: (415) 725–7411

1. Introduction

The general problem we address is that of telling whether constraints remain satisfied as a database changes. Because checking a constraint on a database can be as complex as answering a query, it is desirable to minimize the work done in checking. As an extreme example, we would like to know when any update is guaranteed not to create a violation of a given constraint. In less extreme cases, we look for checks that can be done efficiently and that guarantee no violation occurs, even though sometimes a further, more expensive computation may be needed to check the constraint if the first test fails.

Constraints

A *constraint* is a query whose result is a 0-ary predicate that we call **panic**. If the query produces $\emptyset$ on a given database D, then D is said to *satisfy* the constraint, or the constraint is said to *hold* for D.

Languages for Expressing Constraints

The complexity of checking constraints depends on the language that we use to express the constraint query. Examples of constraint/query languages of interest to us are:

1. Conjunctive queries (Chandra and Merlin [1977]).
2. Nonrecursive datalog, or unions of conjunctive queries (Sagiv and Yannakakis [1981]).
3. Conjunctive queries with arithmetic comparisons (Klug [1988]).
4. Datalog with negation.
5. Recursive datalog, possibly with arithmetic comparisons and/or negation.

Example 1.1: The following constraint is a conjunctive query constraint.

panic `:- emp(E,sales) & emp(E,accounting)`

It states that no employee can be in both the Sales and Accounting departments. Here, we assume *emp* is a predicate representing the traditional Employee-Department relation. We

† Permanent Address: Dept. of CS, Hebrew Univ., Jerusalem, Israel

follow the common Prolog convention that names beginning with a lower-case letter are constants (including predicate names), and names beginning with a capital are variables. □

Example 1.2: The following constraint says that every employee with a salary under 100 must be assigned to a department.

```
panic :- emp(E,D,S) & not dept(D) & S < 100
```

This constraint query is not a conjunctive query, because it has a negated subgoal. It also has a subgoal, $S < 100$, whose predicate is an arithmetic comparison, rather than an abstract symbol. □

Example 1.3: The following constraint query says that every employee must have a salary in the allowed range for the employee's department.

```
panic :- emp(E,D,S) & salRange(D,Low,High) & S < Low
panic :- emp(E,D,S) & salRange(D,Low,High) & S > High
```

Now, the constraint no longer resembles a variety of conjunctive query. However, it is in the class of nonrecursive datalog with arithmetic comparison predicates permitted. That class is the same as finite unions of conjunctive queries, again with arithmetic comparisons permitted. □

Example 1.4: The following constraint says that no employee can be his or her own boss.

```
panic :- boss(E,E)
boss(E,M) :- emp(E,D,S) & manager(D,M)
boss(E,F) :- boss(E,G) & boss(G,F)
```

This constraint query is written in recursive datalog. □

Limits on Available Information

We shall consider three different kinds of problems, depending on how much information we are willing to look at. In general, our goal is to obtain a test using only the allowed information that assures a constraint is not violated.

- The test may assume that certain other constraints are not violated.
- If there is an update to the database involved, the test may assume that no constraints were violated before the update.
- The test must imply that the constraint is satisfied. However, if the test fails, it may or may not be the case that the constraint is satisfied; we need to make a different test involving more information to find out.

Here are the three levels of information that we are interested in:

1. The least information is the constraints alone. This problem corresponds to implication of constraint queries, since the only way we could be sure a constraint C is satisfied without looking at any update or any data is if there were other constraints $C_1, \ldots, C_n$, known to be satisfied, such that whenever C implies **panic**, so does at least one of $C_1, \ldots, C_n$.

2. An intermediate kind of problem is when we are allowed to see both the constraints and the update. This problem has been called "query independent of update" in Elkan [1990], Blakeley and Tompa [1988], Levy and Sagiv [1993]. That is, given a constraint C, known to be satisfied before an update, can we be sure that after a certain update C will continue to hold? More generally, we may also know that certain constraints other than C also hold before the update.

3. The most general problem we shall consider is one in which there are some "local" predicates and some "remote" predicates. We wish to tell whether a constraint C is satisfied after an update, given that it, and perhaps some other constraints, were satisfied before. To make the decision, we are allowed to look not only at C and the update, but also at the data in the relations corresponding to the local predicates. This problem comes up in distributed constraint maintenance, where we would like to avoid the expense of accessing remote data, which in some scenarios is expensive or impossible to access (Gupta and Widom [1992], Gupta and Ullman [1992]).

There is of course the most general problem, where we are allowed to look at all data and constraints, but that is essentially a query evaluation problem.

Correct and Complete Tests

We expect that each test is *correct*, in the sense that whenever it says that the constraint holds, the constraint does hold. Another important property of tests is that they be *complete*, in the sense that whenever the test fails to say that the constraint holds, there is some value the information not accessed by the test may have, such that the constraint does not hold.

2. Applications

We see the theory developed here as having a number of related uses.

1. As we describe it, the problem is to manage a set of constraints $C_1, \ldots, C_n$ on a database D. As D changes, we need to know which if any of the constraints are violated. Generally, we can assume that all constraints hold prior to the most recent change.

2. A related problem concerns active databases, where we have a collection of *rules* of the form "if C holds, then perform action A." We can see such a rule as a constraint **panic** :- C with the action A performed in response to deriving **panic**. We are especially interested in the case where the modifications to the database are the result of the actions of the rules. Unlike (1), we cannot assume that all "constraints" (the conditions in the rules) hold prior to an action, because of the way rules are normally detected and fired (Ceri and Widom [1990, 1991]).

3. Another problem of similar type is view maintenance. We are given an expression defining a view V of a database D, and we want to know when an update to D does *not* affect the value of V. Some conditions for detecting such situations simply have been given by Tompa and Blakely [1988], Blakeley, Coburn, and Larson [1989], and Ceri and Widom [1991].

3. Constraint Subsumption

When we are allowed to look only at the constraints themselves, our only opportunity to take advantage of the information is through subsumption of one constraint by one or more other constraints. If C is a constraint query, and $\mathbf{C} = \{C_1, \ldots, C_n\}$ is a set of constraint queries, we say $\mathbf{C}$ *subsumes* C if whenever C is violated, some C_i in $\mathbf{C}$ is also violated. In that case, there is no need to check C.

Since constraint queries only produce $\{\mathbf{panic}\}$ or $\emptyset$ as a result, subsumption is a special case of containment of programs. Recall that one program P contains another, Q, if on any database the result of P is a superset of the result of Q. We write $Q \subseteq P$ in that case. Then the following is obvious:

Theorem 3.1: Constraint set $\mathbf{C} = \{C_1, \ldots, C_n\}$ subsumes constraint C if and only if, viewed as programs, $C \subseteq C_1 \cup \cdots \cup C_n$. $\Box$

There are many known results about program containment that apply directly to constraint subsumption. For example, if the constraints are conjunctive queries (Chandra and Merlin [1977]) or unions of conjunctive queries (Sagiv and Yannakakis [1981]; these are equivalent to nonrecursive datalog programs), then there is a test for subsumption. In this case, the subsumption test is known to be NP-complete. However, since constraints tend to be short, the exponential complexity of the problem may not present a bar to solution in general.

We can extend conjunctive queries to allow arithmetic comparisons as subgoals (Klug [1988]). The problem is still in NP if we allow the subsuming constraints to be a (possibly recursive) datalog program (Ramakrishnan et al. [1989]). If we allow the subsumed constraint to be a recursive datalog program, while the subsuming constraints are non-recursive datalog, the problem remains decidable (Courcelle [1991]). The complexity of this problem was resolved by Chaudhuri and Vardi [1992], who showed it is complete for doubly exponential time, with some less complex special cases. On the other hand, when both the subsuming and subsumed constraints are recursive datalog, the problem becomes undecidable (Shmueli [1987]).

Reduction of Query Containment to Constraint Subsumption

Since constraint queries have a 0-ary goal predicate, one might wonder if the above cited results are too conservative; i.e., constraint subsumption is easier than query containment. It appears that for any common class of queries, that is not the case. Rather, constraint subsumption is just as hard as the corresponding query containment problem. For powerful query languages, where intermediate predicates are allowed, and several rules may be used (e.g., nonrecursive datalog), it is easy to reduce query containment to the corresponding constraint subsumption problem by adding rules, thus providing a lower bound on the complexity of constraint subsumption. However, when constraints are single conjunctive queries, it may not be clear that the problems are the same. Thus, we state:

Theorem 3.2: Constraint subsumption for conjunctive queries is NP-complete.

Proof: We reduce conjunctive query containment to constraint subsumption. If Q is a conjunctive query of the form h :- B, we rename the predicate of the head h if it appears in the body B. We then "move" the head into the body, creating the conjunctive query

Q' that is **panic** :- h & B. If Q and R are two conjunctive queries, it is easy to check that $Q \subseteq R$ if and only if $Q' \subseteq R'$. □

4. Using the Update

There are similar observations that can be made when we are allowed to use a collection of constraints and an update to determine that another constraint is satisfied. When we start with conjunctive queries as our language, however, the modification to constraints needed to handle an update that is an insertion or deletion takes the problem outside the class of conjunctive queries, into the nonrecursive datalog class. An example will illustrate the point and also indicate how constraints can be modified to account for updates.

Example 4.1: Suppose there are two constraints in an employee database:

C_1: **panic** :- `emp(E,D,S) & not dept(D)`
C_2: **panic** :- `emp(E,D,S) & S > 100`

C_1 is a referential integrity constraint; it says that every employee must be in a department that is mentioned in the *dept* relation. C_2 says that no employee may have a salary greater than 100. We assume these constraints hold before update.

Suppose there is an update in which *toy* is added to the set of departments. We can define a constraint that represents C_1 after the update as

```
dept1(D) :- dept(D)
dept1(toy)
panic :- emp(E,D,S) & not dept1(D)
```

Call this constraint C_3. It is in the language of nonrecursive datalog with negation, even though the constraint C_1 from which it was derived is in the narrower class of conjunctive queries with negation. Then in order to be sure that C_1 has not become violated by the update we need to check $C_3 \subseteq C_1 \cup C_2$. This happens to be the case, and in fact, C_2 is not needed in the containment. The methods of Levy and Sagiv [1993] suffice. □

Example 4.2: Continuing with Example 4.1, suppose we delete the tuple $(jones, shoe, 50)$ from the *emp* relation. Then we need to construct a new predicate *emp*1 that reflects the deletion of this tuple. Here is one way to do so.

```
emp1(E,D,S) :- emp(E,D,S) & E<>jones
emp1(E,D,S) :- emp(E,D,S) & D<>shoe
emp1(E,D,S) :- emp(E,D,S) & S<>50
```

This predicate *emp*1 can substitute for *emp* in either C_1 or C_2 to create new constraints C_4 and C_5 that reflect the situation after this deletion. We then need to check $C_4 \subseteq C_1 \cup C_2$ and $C_5 \subseteq C_1 \cup C_2$. Note that in this construction, conjunctive queries are brought into the class of nonrecursive datalog with arithmetic comparisons. □

5. Using Local Data

The main results of this paper concern the problem of checking constraints given an update and some predicates that are defined to be "local." We assume that for one reason or another (e.g., the expense of querying remote databases) we prefer to check the constraints using only the local data, and we would therefore like to derive the complete local test for our constraints. We show how to derive complete local tests for some important classes of constraints.

For simplicity, we shall focus on conjunctive query constraints (CQC's) of the following form

panic `:-` l `&` r_1 `&...&` r_n `&` c_1 `&...&` c_k

Here, l is the one subgoal with a local predicate (although we can see l as a conjunction of local subgoals). Each of the r_i's is a subgoal with a remote predicate, and each of the c_i's is an arithmetic comparison.

- For simplicity in some situations, we assume that no variable appears twice among l and the r_i's. Rather, multiple occurrences are handled by using distinct variables and equating them by arithmetic equality constraints.
- We assume that the update is the insertion of a tuple into the relation for l.

A Necessary and Sufficient Condition for Containment of CQC's

We first prove a result that applies to general conjunctive queries with arithmetic comparisons, including those with nontrivial heads. If C is a conjunctive query, let $A(C)$ be the set of subgoals of C that are arithmetic comparisons, and let $O(C)$ be the other, "ordinary" subgoals.

Theorem 5.1: Let C_1 and C_2 be CQC's. Then $C_1 \subseteq C_2$ if and only if the following holds. Let H be the set of all containment mappings (mappings from variables to variables that maps head to head and subgoals into subgoals; see Ullman [1989]) from $O(C_2)$ to $O(C_1)$. Then H is nonempty, and $A(C_1)$ logically implies $\vee_{h \text{ in } H}\, h\big(A(C_2)\big)$. □

- Theorem 5.1 generalizes to the containment of C_1 in a union of CQC's in the obvious way. We must include containment mappings from any member of the union to C_1.

Example 5.1: There is in Ullman [1989] a theorem similar to Theorem 5.1 that gives a sufficient but not necessary condition for $C_1 \subseteq C_2$. Example 14.7 from Ullman [1989] gives a counterexample to the version of our Theorem 5.1 in which we are allowed to use only one of the containments mappings in H. Here is Example 14.7 restated as constraints for simplicity (but Theorem 5.1 holds even if the head is not **panic**) and rewritten to avoid repetition of variables in ordinary subgoals.

C_1: **panic** `:- r(U,V) & r(S,T) & U=T & V=S`
C_2: **panic** `:- r(U,V) & U <= V`

Intuitively, C_1 causes panic when the relation for r has a tuple and its reverse, while C_2 causes panic when r has a tuple with first component no greater than the second component. Clearly, whenever r has both (a, b) and (b, a), there will be in r a tuple with first component less than or equal to the second. Thus, we expect $C_1 \subseteq C_2$, but the test of Ullman [1989] does not indicate so.

To apply Theorem 5.1, note that there are two containment mappings from the ordinary subgoals of C_2 to those of C_1:

1. $h(U) = U$; $h(V) = V$.
2. $g(U) = S$; $g(V) = T$.

Theorem 5.1 thus tells us we must verify $A(C_1) \Rightarrow h\big(A(C_2)\big) \lor g\big(A(C_2)\big)$. That is,

$$U = T \land V = S \Rightarrow U \leq V \lor S \leq T$$

The above simplifies to $U \leq V \lor V \leq U$, which is true assuming that $\leq$ is a total order. □

Comparison With Klug's Approach

The reader should compare the approach of Theorem 5.1, where we need to consider all containment mappings, to the approach of Klug [1988], where all orders consistent with the arithmetic constraints are considered. Both approaches can be exponential in the worst case, and in fact the general problem is NP-complete (van der Mayden [1992]).

Klug's approach in the worst case requires an exponential number of tests, each of which could take exponential time. Ours creates one test that could in the worst case be exponential in the size of C_2, but the test for satisfaction of the implication is exponential only in the number of variables, that is, in the size of C_1.

However, for constraint checking, it is likely that the conjunctive queries involved will have few duplicate predicates, and these predicates and their multiplicities will be the same in both queries. (The material following shortly on reduction-based tests justifies this position.) Moreover, as we saw in Example 5.1, there may be logical simplifications that facilitate the checking of the implication. Thus, we expect our approach to perform better in real cases. In Section 6, we see applications where the approach embodied in Theorem 5.1 appears to be necessary to handle some special cases of CQC's.

Instantiating Local Predicates

In order to handle conjunctive queries with local predicates and available local data, it is useful to develop a notation for instantiating the local predicate in all possible ways. If t is a tuple that could be in the relation for predicate l, and C is a CQC of the form described in the beginning of Section 5, then $\text{RED}(t, l, C)$, the *reduction* of C by t in l, is obtained by substituting the components of t for the corresponding variables in the arguments of l, and then eliminating l.

Example 5.2: We shall now introduce an important example, which we call *forbidden intervals.*

C: **panic** `:- l(X,Y) & r(Z) & X<=Z & Z<=Y`

That is, each pair in the local relation can be thought of as the ends of an interval which no Z in the remote relation may occupy.

Suppose that the relation for l has pairs $(3, 6)$ and $(5, 10)$, and we insert into l the tuple $(4, 8)$. Then:

$\text{RED}((3, 6), l, C)$ = `r(Z) & 3<=Z & Z<=6.`
$\text{RED}((5, 10), l, C)$ = `r(Z) & 5<=Z & Z<=10.`
$\text{RED}((4, 8), l, C)$ = `r(Z) & 4<=Z & Z<=8.`

Note that $\text{RED}((4,8),l,C) \subseteq \text{RED}((3,6),l,C) \cup \text{RED}(5,10),l,C)$. This observation tells us that when the stated insertion occurs, we need not fear that C is violated, if it was not previously violated. That is, the presence of the tuples $(3,6)$ and $(5,10)$ in l together assure us that the interval from 4 to 8 is clear of elements in the remote relation for r. □

- Example 5.2 points out that when CQC's have arithmetic comparisons, there may be a containment of one CQC C in the union of several, yet not have the union of C in any one CQC of the union. The same cannot happen if there are no arithmetic comparisons (Sagiv and Yannakakis [1981]). The need to consider containment of a CQC in several CQC's is the reason that the results of Gupta and Ullman [1992] or Gupta and Widom [1993] cannot be extended.

The following theorem formalizes the complete local test suggested by Example 5.2 for preservation of a single constraint.

Theorem 5.2: Let C be a CQC and let t be a tuple inserted into the local relation L for predicate l. Assume C holds before the update. Then the complete local test for guaranteeing that C holds after the update is whether $\text{RED}(t,l,C) \subseteq \cup_{s \text{ in } L} \ \text{RED}(s,l,C)$.

Proof: Straightforward argument building on the results of Nicolas [1982]. □

- Theorem 5.2 extends to the case where several constraints are assumed to hold prior to the update. It also extends to the case where several tuples are inserted at once.

Constructing Complete Local Tests for CQC's

Theorems 5.1 (extended to cover containment of a CQC in a union of CQC's) and 5.2 suggest an algorithm to test whether a CQC C is preserved by a local update. If t is the inserted tuple, and L is the local relation for predicate l, let H_s, for s in L, be the set of containment mappings from $\text{RED}(s,l,C)$ to $\text{RED}(t,l,C)$. For any tuple u, let C_u be $A\big(\text{RED}(u,l,C)\big)$, that is, the arithmetic comparisons in the reduction of C by u in l. Then we need to test whether C_t logically implies $\vee_{s \text{ in } L, h \text{ in } H_s} \ h(C_s)$.

Arithmetic Free CQC's

If there are no arithmetic comparisons in a CQC, the test of Theorem 5.2 can be simplified. There are two key differences when arithmetic is absent:

1. Whenever $\text{RED}(t,l,C)$ is contained in the union of reductions over all tuples in L, then $\text{RED}(t,l,C)$ is contained in one such reduction.

2. The containment of $\text{RED}(t,l,C)$ in $\text{RED}(s,l,C)$ depends not on the values of the components of the tuples t and s, but only on their patterns of equality and inequality.

Thus, while the test in general could be exponential in the size of the local relation L and also in the CQC, for the arithmetic-free case the test is exponential only in the size of the CQC.

Theorem 5.3: In time at most exponential in the size of an arithmetic-free CQC it is possible to construct an expression of relational algebra whose nonemptiness is the complete local test for preservation of the CQC after an insertion to the local relation.

Proof: The main ideas are as follows. Suppose t is the inserted tuple, and C is an arithmetic-free CQC with local relation L for predicate l. Let μ be a tuple of variables of the length appropriate for L. Then we need only ask if there is a containment mapping from $\text{RED}(\mu, l, C)$ to $\text{RED}(t, l, C)$. If there are one or more, then each containment mapping provides a set of constraints on the variables in μ. These can easily be translated into an algebraic expression on L. □

Example 5.3: The ideas should become clearer from the following example. Consider the arithmetic-free CQC

C_1: **panic** `:- l(X,Y,Y) & r(Y,Z,X)`

Let $\mu = (A, B, C)$. Then $\text{RED}(\mu, l, C_1)$ is `r(B,Z,A)`. Note that variables B and C were equated in μ by the substitution into the subgoal $l(X, Y, Y)$.

Now, consider inserted tuple $t = (a, b, c)$. $\text{RED}(t, l, C_1)$ does not exist, because $b \neq c$ and there is thus no way to unify $l(t)$ with $l(X, Y, Y)$. Thus, there is no condition under which the insertion of t could invalidate C_1; i.e., the complete local test is "true".

Next consider the inserted tuple $s = (a, b, b)$. Now, $\text{RED}(s, l, C_1)$ is `r(b,Z,a)`. There is one containment mapping from $\text{RED}(\mu, l, C_1)$ to $\text{RED}(s, l, C_1)$, namely $h(A) = a$; $h(Z) = Z$; $h(B) = b$. This containment mapping says that the first two components of μ must be a and b, respectively. Moreover, the unification of μ with $l(X, Y, Y)$ required that the second and third components of μ be equal; i.e., $B = C$. Expressed in relational algebra on the relation L of which μ is an abstract tuple, this expression is $\sigma_{\#1=a \wedge \#2=b \wedge \#3=b}(L)$. That is, when we insert a tuple (a, b, b) into L, the complete local test is whether this tuple already exists in L, not a very useful test, but one that technically should be made. □

6. Special Cases Solvable by Recursive Datalog

The problem of CQC's including arithmetic comparisons appears quite hard in general. However, we are able to solve two interesting cases, showing a way to express the complete local test as a (recursive) datalog program with arithmetic.

When There is Only One Remote Variable

The first case is when there is only one *remote* variable in the CQC; that is, only one variable that does not appear in the local subgoal. The forbidden intervals problem of Example 5.2 is typical.

Example 6.1: The CQC

C: **panic** `:- l(X,Y) & r(Z) & X<=Z & Z<=Y`

Has only one remote variable, Z. Informally it is violated whenever there is a remote value Z that lies in some interval $[X, Y]$ defined by a pair (X, Y) in the relation L for l. Thus, if we insert a tuple (a, b) into L we can be sure C continues to hold if and only if the union of all the intervals (X, Y) in L covers $[a, b]$. Thus, inclusion of interval $[a, b]$ into the union of all the intervals of L is a complete local test.

- We can observe therefore that, unlike Theorem 5.3, this constraint C does not have a complete local test that is an expression of relational algebra. The intuition is that if there were such an expression, there would be a bound k, derivable from the expression, such that at most k different tuples of L are "looked at" in determining the presence of any one tuple in the result. However, we can then concoct an example where it takes $k+1$ tuples to "cover" the inserted tuple, proving that the hypothetical expression cannot be correct.

We can write a recursive datalog program with arithmetic to construct maximal intervals from the tuples in L. That is

```
interval(X,Y) :- l(X,Y)
interval(X,Y) :- interval(X,W) & interval(Z,Y) & Z <= W
ok(A,B) :- interval(X,Y) & X <= A & B <= Y
```

That is, we combine overlapping intervals into one interval that includes them both, until we have the longest possible intervals constructed from the given intervals. Finally, given an inserted tuple (a, b), we need only determine whether $ok(a, b)$ is true. This is the complete local test. □

In the paper, we shall prove the following theorem, whose proof follows the ideas of Example 6.1.

Theorem 6.1: For any CQC with only one remote variable, we can construct a datalog program with arithmetic to serve as a complete local test for the CQC. □

Independently Constrained Queries

The second case of interest is when all remote variables are limited, if they are limited at all, by comparisons to a constant or a local variable, rather than to another remote variable. Formally, we define a CQC C to be *independently constrained* (an ICQ) if:

1. There are no inequality ($\neq$) comparisons in C.
2. If t is a tuple of constants, then $\text{RED}(t, l, C)$ has no instance of a subgoal of the form $X < Y$, $X \leq Y$, $X > Y$, or $X \geq Y$. That is, all greater/less-than comparisons involve at most one remote variable.

Theorem 6.2: For any ICQ we can construct a recursive datalog program with arithmetic to serve as a complete local test for the CQC. □

Bibliography

Blakeley, J. A., N. Coburn, and P.-A. Larson [1989]. "Updating derived relations: detecting irrelevant and autonomously computable updates," *ACM Trans. on Database Systems* **14**:3, pp. 369–400.

Ceri, S. and J. Widom [1990]. "Deriving production rules for constraint maintainence," *Proc. International Conference on Very Large Data Bases*, pp. 566–577.

Ceri, S. and J. Widom [1991]. "Deriving production rules for incremental view maintainence," *Proc. International Conference on Very Large Data Bases*, pp. 577–589.

Chandra, A. K. and P. M. Merlin [1977]. "Optimal implementation of conjunctive queries in relational databases," *Proc. Ninth Annual ACM Symposium on the Theory of Computing,* pp. 77–90.

Chaudhuri, S. and M. Y. Vardi [1992]. "On the equivalence of datalog programs," *Proc. Eleventh ACM Symposium on Principles of Database Systems*, pp. 55–66.

Courcelle, B. [1991]. "Recursive queries and context-free graph grammars," *Theor. CS* **78**, pp. 217–244.

Elkan, C. [1990]. "Independence of logic database queries and updates," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 154–160.

Gupta, A. and J. D. Ullman [1992]. "Generalizing conjunctive query containment for view maintenence and integrity constraint verification," *Joint Intl. Conf. on Logic Programming, Workshop on Deductive Databases.*

Gupta, A. and J. Widom [1993]. "Local verification of global integrity constraints in distributed databases," *ACM SIGMOD International Conf. on Management of Data*, pp. 49–58.

Klug, A. [1988]. "On conjunctive queries containing inequalities," *J. ACM* **35**:1, pp. 146–160.

Levy, A. Y. and Y. Sagiv [1993]. "Queries independent of update," *Proc. International Conference on Very Large Data Bases,* pp. 171–181.

Nicolas, J.-M. [1982]. "Logic for improving integrity checking in relational databases," *Acta Informatica* **18**:3, pp. 227–253.

Ramakrishnan, R., Y. Sagiv, J. D. Ullman, and M. Y. Vardi [1989]. "Proof tree transformation theorems and their applications," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 172–181.

Sagiv, Y. and M. Yannakakis [1981]. "Equivalence among relational expressions with the union and difference operators," *J. ACM* **27**:4, pp. 633–655.

Shmueli, O. [1987]. "Decidability and expressiveness aspects of logic queries," *Proc. Sixth ACM Symposium on Principles of Database Systems*, pp. 237–249.

Tompa, F. W. and J. A. Blakeley [1988]. "Maintaining materialized views without accessing base data," *Inform. Systems* **13**:4, pp. 393–406.

Ullman, J. D. [1989a]. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*, Computer Science Press, New York.

van der Meyden, R. [1992]. "The complexity of querying indefinite data about linearly ordered domains," *Proc. Eleventh ACM Symposium on Principles of Database Systems*, pp. 331–345.